Annotation Inference for the Safety Certification of Automatically Generated Code

Ewen Denney
USRA/RIACS, NASA Ames
edenney@email.arc.nasa.gov

Bernd Fischer
ECS, University of Southampton
B.Fischer@ecs.soton.ac.uk

Abstract

Code generators for realistic application domains are not directly verifiable in practice. Certifiable code generation is an alternative approach, where the the generator is extended to generate logical annotations (i.e., pre- and postconditions and loop invariants) along with the programs, allowing fully automated program safety proofs. However, it is difficult to implement and maintain because it requires access to the generator sources, and because the annotations are cross-cutting concerns, both on the object-level (i.e., in the generated code) and on the meta-level (i.e., in the generator).

We describe a generic post-generation annotation inference algorithm to circumvent these problems. It exploits the fact that the output of a code generator is highly idiomatic, so that patterns can be used to describe all code constructs that require annotations, in a way which is largely independent of the code generator. The algorithm then uses techniques similar to aspect-oriented programming to add the annotations to the generated code. The generic algorithm is implemented and instantiated for two generators; the instantiations are applied successfully to fully automatically certify initialization safety for the generated programs.

1 Introduction

Automated code generation is an enabling technology for model-based software development and has significant potential to improve the entire software development process. It promises many benefits, including higher productivity, reduced turn-around times, increased portability, and elimination of manual coding errors. However, the key to realizing these benefits is of course generator correctness—nothing is gained from replacing manual coding errors with automatic coding errors.

Several ideas have been explored to ensure generator correctness, but none has been entirely successful yet. Approaches based on "correct-by-construction" techniques like deductive synthesis [23] or refinement [22] remain difficult to implement and to scale up, and have not found widespread application. The direct verification of generators is still a challenge for existing verification techniques and remains similarly elusive. Usually, generators are thus validated by testing only, but this quickly becomes excessive and cannot guarantee the same level of assurance.

Our work follows an alternative approach that is based on the observation that the correctness of the generator is irrelevant if instead the correctness of the generated programs is shown individually. However, rather than showing full correctness of the generated programs as envisioned for a verifying compiler [15], we follow the more pragmatic certifying compiler [19] used in the proof carrying code (PCC) approach and focus on the Hoare-style certification of specific safety properties.

Still, this leaves us with the task to construct the appropriate logical annotations (i.e., pre- and postconditions and loop invariants), due to their central role in Hoare-style certification. In previous work [5, 6, 7, 8, 25] we have thus developed and evaluated an approach to certifiable program generation in which the code generator itself is extended in such a way that it produces all necessary annotations together with the code. This is achieved by embedding annotation templates into the code templates, which are then instantiated and refined in parallel by the generator.

We have successfully used this approach to certify a variety of safety properties for code generated by the AUTOBAYES [11] and AUTOFILTER [26] systems. However, it has two major disadvantages. First, it is difficult to implement and to maintain: the developers first need to analyze the generated code in order to identify the location and structure of the required annotations, then identify the templates that produce the respective code fragments, and finally formulate and integrate appropriate annotation templates. This is compounded by the fact that annotations are cross-cutting concerns, both on the object-level (i.e., the generated program) and the meta-level (i.e., the generator). Second, it requires access to the existing sources: the devel-

opers need to modify the code generator in order to integrate the annotation generation. However, sources are often not accessible, in particular for commercial generators.

In this paper we describe an alternative approach that uses a generic post-generation annotation inference algorithm to circumvent these problems. It is based on three fundamental insights: (*i*) the problem of certifying code can be split into two phases: an untrusted annotation generation where the creative insights are made, and a simpler trusted phase where verification conditions are generated and proven (*ii*) although "eureka" insights are required in general, in practice they are rarely required because certain common cases typically arise, and (*iii*) code patterns can be used to describe these cases.

The algorithm can run completely separately from the code generator because it uses techniques similar to aspect-oriented programming to add the annotations to the generated code: the patterns correspond to (static) point-cut descriptors, while the introduced annotations correspond to advice. It thus concentrates annotation generation in one location but, even more importantly, leaves the generator unchanged by exploiting the idiomatic structure of automatically generated code (i.e., the fact that it only constitutes a limited subset of all possible programs).

The main contribution of this paper is a general approach to extending code generators with a certification capability, which has been validated on two code generators. We build on previous work on certifiable code generation to develop a *generic* certification system for code generators. However, our focus in this paper is on the annotation inference, rather than the subsequent generation and proof of verification conditions.

2 Background

Idiomatic Code Automatic code generators derive lower-level code from higher-level, declarative specifications. Approaches range from deductive synthesis [23] to template meta-programming [4] but for our purposes neither the specific approach nor the specification language matter, and we build on a template-based approach. What *does* matter, however, is the fact that an automatic code generator usually generates highly *idiomatic code*. Intuitively, idiomatic code exhibits some regular structure beyond the syntax of the programming language and uses similar constructions for similar problems. Manually written code already tends to be idiomatic, but the applied idioms vary with the programmer. Automatic generators eliminate this variability because they essentially derive code by combining a finite number of building blocks—in our case, templates. For example, AUTOFILTER only uses three templates to initialize a matrix, resulting in either straight-line code or one of two doubly-nested loop versions (cf. Figure 1)

```
A[1,1] := a_{1,1};   for i := 1 to n do     for i := 1 to n do
...                    for j := 1 to m do    for j := 1 to m do
A[1,m] := a_{1,m};     B[i,j] := b;            if i=j then
A[2,1] := a_{2,1};                               C[i,j] := c
...                                            else
A[n,m] := a_{n,m};                               C[i,j] := c';
```

Figure 1. Idiomatic matrix initializations

The idioms are essential to our approach because they (rather than the templates) determine the interface between the code generator and the inference algorithm. Note that the idioms can be recognized even without knowing the templates that produced the code, which allows us to apply our technique to black-box generators as well. However, identifying and formalizing the necessary idioms remains a manual step in the process.

Safety Certification The purpose of safety certification is to demonstrate that a program does not violate certain conditions during its execution. A *safety property* is an exact characterization of these conditions based on the operational semantics of the language. A *safety policy* is a set of Hoare rules designed to show that safe programs satisfy the safety property of interest. The rules can be formalized using the usual Hoare triples $P\ \{c\}\ Q$, i.e., if the condition P holds before and the command c terminates, then Q holds afterwards (see [18] for more information about Hoare-style program proofs).

For each notion of safety the appropriate safety property and corresponding policy must be formulated. This is usually straightforward; in particular, a safety policy can be constructed systematically by instantiating a generic rule set that is derived from the standard rules of the Hoare calculus [5]. The basic idea is to extend the standard environment of program variables with a "shadow" environment of safety variables which record safety information related to the corresponding program variables. The rules are then responsible for maintaining this environment and producing the appropriate safety obligations. This is done using a family of *safety substitutions* that are added to the normal substitutions, and a family of *safety predicates* that are added to the calculated weakest preconditions (WPCs). Safety certification then starts with the postcondition *true* and computes the weakest safety precondition (WSPC), i.e., the WPC together with all applied safety predicates and safety substitutions. If the program is safe then the WSPC will be provable without any assumptions, i.e., $true\ \{c\}\ true$ is derivable.

We have defined several safety properties and implemented the corresponding safety policies in AUTOBAYES and AUTOFILTER. Here, we focus on the initialization safety policy, which ensures that each variable or individual array element has been explicitly assigned a value before it

$$(assign)\quad \frac{}{Q[e/x, \text{INIT}/x_{\text{init}}] \wedge safe_{\text{init}}(e)\ \{x := e\}\ Q}$$

$$(update)\quad \frac{}{Q[upd(x, e_1, e_2)/x, upd(x_{init}, e_1, \text{INIT})/x_{\text{init}}] \wedge safe_{\text{init}}(e_1) \wedge safe_{\text{init}}(e_2)\ \{x\,[e_1]\ := e_2\}\ Q}$$

$$(if)\quad \frac{P_1\ \{c_1\}\ Q \quad P_2\ \{c_2\}\ Q}{(b \Rightarrow P_1) \wedge (\neg b \Rightarrow P_2) \wedge safe_{init}(b)\ \{\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\}\ Q}$$

$$(while)\quad \frac{P\ \{c\}\ I \quad I \wedge b \Rightarrow P \quad I \wedge \neg b \Rightarrow Q}{I \wedge safe_{\text{init}}(b)\ \{\textbf{while}\ b\ \textbf{inv}\ I\ \textbf{do}\ c\}\ Q}$$

$$(for)\quad \frac{P\ \{c\}\ I[i+1/i] \quad I[\text{INIT}/i_{init}] \wedge e_1 \le i \le e_2 \Rightarrow P \quad I[e_2+1/i] \Rightarrow Q}{I[e_1/i] \wedge safe_{\text{init}}(e_1) \wedge safe_{\text{init}}(e_2)\ \{\textbf{for}\ i := e_1\ \textbf{to}\ e_2\ \textbf{inv}\ I\ \textbf{do}\ c\}\ Q}$$

$$(comp)\quad \frac{P\ \{c_1\}\ R \quad R\ \{c_2\}\ Q}{P\ \{c_1\ ;\ c_2\}\ Q} \qquad (skip)\quad \frac{}{Q\ \{\textbf{skip}\}\ Q} \qquad (assert)\quad \frac{P' \Rightarrow P \quad P\ \{c\}\ Q' \quad Q' \Rightarrow Q}{P'\ \{\textbf{pre}\ P'\ c\ \textbf{post}\ Q'\}\ Q}$$

Figure 2. Proof rules for initialization safety

is used. The safety environment consists of shadow variables x_{init} that contain the value INIT after the variable x has been assigned a value. Arrays are represented by shadow arrays to capture the status of the individual elements. Figure 2 shows the rules of the policy. Only statements accessing assigning a value to a location affect the value of a shadow variable (cf. the *assign*-, *update*-, and *for*-rules). However, all rules also produce the appropriate safety predicates $safe_{\text{init}}(e)$ for all immediate subexpressions e of the statements. Since the safety property defines an expression to be safe if all corresponding shadow variables have the value INIT, $safe_{\text{init}}(x\,[i]\,)$ for example simply translates to $i_{\text{init}} = \text{INIT} \wedge (x_{\text{init}}\,[i]\,) = \text{INIT}$.

VC Processing and Annotations As usual in Hoare-style verification, a verification condition generator (VCG) traverses the annotated code and applies the rules of the calculus to produce verification conditions (VCs). These are then simplified, completed by an axiomatization of the background theory and passed to an off-the-shelf automated theorem prover (ATP). If all VCs are proven, the program is safe wrt. safety property. Note that the ATP has no access to the program internals; hence, all pertinent information must be taken from the annotations, which become part of the VCs. For full functional verification, annotations are thus usually very detailed and, consequently, annotation inference remains intractable for this case. For safety certification, on the other hand, the Hoare-rules have already more internal structure and the safety predicates are regular and relatively small, so that the required annotations are a lot simpler. In addition, the targeted safety property and policy are known at annotation inference time, which eliminates the need for any logical reasoning in the style of the early inference approaches [24].

3 A Worked Example

Figure 3(a) shows a simple example program that initializes two vectors A and B of size N with given but irrelevant values a_i and b (cf. lines 2.1–2.n and 3.1–3.2, resp.) and then computes and returns the sums s and t of their respective elements as well as their dot-product d. It is derived from and representative of the code generated by AUTOFILTER; in particular it shows the same overall structure—a series of variable definitions followed by a loop with variable uses. AUTOFILTER's target language is a simple imperative language with basic control constructs (i.e., **if** and **for**) and numeric scalars and arrays as the only datatypes. However, the language also supports domain-specific operations on entire vectors and matrices like matrix multiplication or assignment, although these are not used in the example shown in Figure 3.

Intuitively, the certification of initialization safety requires that the logical annotations entail at each use of a variable x that the corresponding shadow variable x_{init} has the value INIT. In particular, we need an invariant for the loop at line 5.1 that ensures this for s, t, and d, as well as for the entire arrays A and B.

The first step of the inference algorithm is to scan the program for relevant variables; for initialization safety all variables that are used on the right-hand side of assignments (more precisely, in *rvar*-positions) are relevant, but here we will restrict our attention to the two array variables A and B, starting with B which is used in lines 5.3 and 5.4. Both uses are abstracted into *use*(B), cf. Figure 3(b). The algorithm then follows all control flow paths backwards from the uses until it encounters either a cycle or a definition for the variable. Paths that do not end in a definition are discarded and the remaining paths are traversed node by node, while the

	(a)	(b)	(c)	(d)	(e)
1.1	**const** N := n;	**const** N := n;	**const** N := n;	*block*;	**const** N := n;
1.2	**var** i,s,t,d;	**var** i,s,t,d;	**var** i,s,t,d;		**var** i,s,t,d;
1.3	**var** A[1:N],B[1:N];	**var** A[1:N],B[1:N];	**var** A[1:N],B[1:N];		**var** A[1:N],B[1:N];
2.1	A[1] := a_1;	A[1] := a_1;	A[1] := a_1;	*def*(A[1:N]);	A[1] := a_1;
	...	...	...		...
2.n	A[n] := a_n;	A[n] := a_n;	A[n] := a_n;		A[n] := a_n; **post** $\forall j \in \{1:n\} \cdot A_{\text{init}}[j] = \text{INIT}$
3.1	**for** i := 1 **to** N **do**	*def*(B[1:N]);	**for** i := 1 **to** N **inv** $\forall j \in \{1:i-1\} \cdot B_{\text{init}}[j] = \text{INIT}$ **do**	*barrier*;	**for** i := 1 **to** N **inv** $\forall j \in \{1:n\} \cdot A_{\text{init}}[j] = \text{INIT} \wedge \forall j \in \{1:i-1\} \cdot B_{\text{init}}[j] = \text{INIT}$ **do**
3.2	B[i] := b;		B[i] := b; **post** $\forall j \in \{1:N\} \cdot B_{\text{init}}[j] = \text{INIT}$		B[i] := b; **post** $\forall j \in \{1:n\} \cdot A_{\text{init}}[j] = \text{INIT} \wedge \forall j \in \{1:N\} \cdot B_{\text{init}}[j] = \text{INIT}$
4.1	s := 0;	s := 0;	s := 0;	*block*;	s := 0;
4.2	t := 0;	t := 0;	t := 0;		t := 0;
4.3	d := 0;	d := 0;	d := 0;		d := 0;
5.1	**for** i := 1 **to** N **do**	**for** i := 1 **to** N **do**	**for** i := 1 **to** N **inv** $\forall j \in \{1:N\} \cdot B_{\text{init}}[j] = \text{INIT}$ **do**	**for** i := 1 **to** N **do**	**for** i := 1 **to** N **inv** $\forall j \in \{1:n\} \cdot A_{\text{init}}[j] = \text{INIT} \wedge \forall j \in \{1:N\} \cdot B_{\text{init}}[j] = \text{INIT} \wedge s_{\text{init}} = t_{\text{init}} = d_{\text{init}} = \text{INIT}$ **do**
5.2	s := s+A[i];	t := t+A[i];	s := s+A[i];	*use*(A);	s := s+A[i];
5.3	t := t+B[i];	*use*(B);	t := t+B[i];	*block*;	t := t+B[i];
5.4	d := d+A[i]*B[i];	*use*(B);	d := d+A[i]*B[i];	*use*(A);	d := d+A[i]*B[i]; **post** $s_{\text{init}} = t_{\text{init}} = d_{\text{init}} = \text{INIT}$
6	**return** s,t,d;	**return** s,t,d;	**return** s,t,d;	*block*;	**return** s,t,d;

Figure 3. (a) Original program (b) Abstraction for B (c) Annotations for B (d) Abstraction for A (using *block*-patterns) (e) Fully annotated program.

annotations are added as required.

Here, the only assignment to B is in line 3.2; however, this is not the entire definition—the algorithm needs to identify the **for**-loop (lines 3.1-3.2) as the definition for the entire array B and abstract it into the definition node *def*(B[1:N]). The path search then starts at line 5.4 and goes straight back up to the **for**-loop at line 5.1, where it splits. One branch comes in from the bottom of the loop-body but this immediately leads to a cycle and is therefore discarded. The other branch continues through lines 4.1–4.3 and terminates at the definition node at line 3.1. Since all branches have been exhausted, there is only one path along which annotations need to be added. The annotation process starts with the use and proceeds towards the definition terminating the path. The form of all annotations is fully determined by the known syntactic structure of the definition and by the safety property. Since the definition is a loop, in this case, it needs a loop invariant, as well as a postcondition. Since the safety property is initialization safety, both invariant and postcondition need to formalize that the shadow variable B_{init} corresponding to the current array variable B records the value INIT for the already initialized entries. Note that the different upper bounds for the quantifiers can both be constructed from the loop. The postcondition is then pulled along the remaining path, i.e., added to all nodes that require it. Every node needs to be inspected, but in this case only the **for**-loop at line 5.1 requires an invariant. Figure 3(c) shows the result of this pass.

The next pass (cf. Figure 3(d)) adds the annotations for A. As before, its two uses in lines 5.2 and 5.4 are abstracted. A is initialized using a different idiom—a sequence of assignments, cf. lines 2.1–2.n—but this is again collapsed into a *def*-node; here, the initialized range is taken from the first and last assignment, respectively. The program is then collapsed further by the introduction of *barrier*- and *block*-nodes. These represent areas that do not need to be explored because they cannot contain relevant definitions, thus substantially reducing the number of paths. However, the *barrier*-nodes must be re-expanded during the path traversal phase because they require annotations (cf. line 3.1) while *block*-nodes remain opaque. Except for this special handling, the algorithm proceeds as before, and Figure 3(e) shows the fully annotated example program.

4 Inference Algorithm

The previous section shows that the set of idiomatic coding patterns which are used is the key knowledge that drives the annotation construction. However, this is not a general program understanding problem: we are not concerned with identifying general-purpose coding patterns and clichés [21] but only the relevant definitions and uses. These are specific to the given safety property, but the algorithm remains the same for each policy. In the case of initialization safety, the definitions are the different initialization blocks as shown in Figure 1, while the uses are statements which read a variable (i.e., contain an *rvar*).

```
global  SP:Property;
        P :AST;
prov ann_prog() =
var patterns: list Pattern;
    var      : Id;
    uses     : list Position;
    use      : Position;
    cfg      : CFG;
    path     : Path
begin
  patterns := get_patterns(SP);
  foreach (var,uses) in compute_hotvars() do
    cfg := compute_cfg(patterns, var);
    foreach use in uses do
      foreach path in compute_paths(cfg, use) do
        ann_path(path);
end
```

Figure 4. Top-level Algorithm

The aim of the inference algorithm is to "get information from definitions to the uses", i.e., to annotate the program in such a way that the VCG will have the necessary information to show the program safe as it works its way back through the program. For each variable, the algorithm first computes each control flow path back from a use to a definition and then traverses the paths, annotating the definitions and all intermediate nodes that otherwise constitute barriers to the information flow. The following subsections describe the various concepts and components.

As in the certifiable code generation approach, we still split the certification problem into two phases—first, we infer the logical annotations required to prove the code safe; second, we apply the standard machinery (i.e., VCG and ATP) to prove that the code complies with the annotations. As consequence of splitting annotation *inference* from checking annotation *compliance* we do not need to carry out any logical analysis during annotation. Moreover, the inference algorithm remains an untrusted component in the sense of the PCC model.

4.1 Top-level Algorithm Structure

The top-level structure of the algorithm (cf. Figure 4) closely follows the outline above. The safety property `SP` and the abstract syntax tree of the program `P` are used by all functions and given as global variables. The overall result is returned by side-effects on `P`. `ann_prog` first accesses the property-specific patterns for definitions, uses and barriers. It then further reduces the inference efforts by limiting the analysis to certain program hot spots which are determined by the so-called "hot variables" described in the next section.

4.2 Hot Variable Identification

Proving a program safe requires annotations at the points where the VCG needs essential information about the definition of certain key variables. To see why some uses of variables are more critical than others, consider how a VCG processes a program to generate VCs. The VCG works back through the program, gradually constructing a WPC and generating safety obligations whenever required by the use of a variable. The safety obligations will ultimately be discharged in the context of safety substitutions that accumulate earlier in the program. If something is missing from that context, it must be supplemented by an annotation. Therefore, to figure out which annotations are required, we need to know at which points variables are used with "missing" information: we need a notion of availability.

Informally, we say that a variable is *available* at some point in a program (wrt. a safety property) if it is within reach of its definition. For example, immediately after a scalar assignment, the assigned variable is available but it becomes unavailable if there is an intervening loop. We say that a variable use is *hot* if it unavailable, and call a variable a "hot variable" (hotvar, for short) if at least one of its uses is hot.

The algorithm can then pass through the program before the annotation phase, and collect hotvars and uses, since these are the only variables for which it needs to construct annotations. Limiting the analysis in this way is a crucial optimization to cut down the number of graphs to be constructed (cf. Section 4.4).

The function `compute_hotvars` maintains a list of available variables, initially set to empty, and scans forward through the program, deciding for each statement (and the given property) how it affects the availability of the variables. For example, we assume that scalar assignments add to the available variables, but array assignments do not: because arrays are typically accessed indirectly using loops and variable indices, all uses should be treated as hot. For each statement that matches the policy-specific use pattern, the algorithm also checks if the used variable is available; if it is not, that use is tagged as being hot.

Note that the hotvars are computed before the pattern

$$
\begin{array}{rcll}
P & ::= & x & x \in X \\
 & | & f(P_1, \ldots, P_n) & f \in \Sigma \\
 & | & \_ \mid P? \mid P* \mid P+ & \\
 & | & P_1 \,||\, P_2 \mid P_1 \,;\, P_2 & \\
 & | & P_1 \in P_2 \mid P_1 \notin P_2 &
\end{array}
$$

Figure 5. Pattern Grammar

analysis, and in order to minimize the work in that and subsequent stages. The hot variables are therefore approximated conservatively, i.e., we err on the side of designating uses (and could even treat all uses) as hot.

4.3 Patterns and Pattern Matching

The algorithm uses patterns to capture the idiomatic code structures and pattern matching to find the corresponding code locations. Each pattern specifies a class of fragments that are treated similarly by the algorithm, e.g., because they require a similar annotation.

The pattern language is essentially a tree-based regular expression language similar to XML-based languages like XPath [3]; Figure 5 shows its grammar. The language supports matching of tree literals $f(P_1, \ldots P_n)$ (if the signature Σ is given by the programming language to be analyzed, we will also use its concrete syntax to formulate example patterns), wildcards (_) and the usual regular operators for optional (?), list (*) and non-empty list (+) patterns, as well as alternation (||) and concatenation (;) operators. It also supports matching at arbitrary subterm positions (i.e., $P_1 \in P_2$ matches all terms that match P_2 and have at last one subterm that matches P_1; similarly, $P_1 \notin P_2$ matches all terms that match P_2 and have no subterm that matches P_1). Matching arbitrarily nested terms of the form $f(\cdots f(x) \cdots)$ is not required for our purposes.

However, the main difference to XPath and similar languages is that we use meta-variable patterns x to introduce a limited degree of context dependency. Like a wildcard, an uninstantiated meta-variable matches any term but, unlike a wildcard, it becomes instantiated with the matched term and thus subsequently only in other instances of the instantiated pattern. For example, the pattern (_[_] := _)+ matches the entire statement list `A[1]:=1;A[2]:=2;B[1]:=1` while the pattern (x[_] := _)+ matches, after the instantiation of x with `A` on the first statement, only the following second assignment to `A` but not the final assignment to `B`. Further context-dependencies are introduced by multiple occurrences of the same meta-variable in a pattern. Hence, the pattern **for** x := _ **to** _ **inv** _ **do** _[x, x] := _ can be used to identify loops that access only the diagonal elements of any matrix.

The match procedure traverses terms first top-down and then left-to-right over the direct subterms. Meta-variables are instantiated eagerly (i.e., as close to the root as possible) but instantiations are undone if the enclosing pattern fails later on. List patterns follow the usual "longest match" strategy used in almost all traditional regular expression matchers. The match procedure returns as result a set of (*Position* $\times$ $\mathbb{N}$ $\times$ *Substitution*)-triples where the first two arguments are the root position and length of the match of the top-level pattern.

4.4 Abstracted Control Flow Graphs

The algorithm follows the control flow paths from variable use nodes backwards to all corresponding definitions and annotates the statements along these paths as required (see the next two sections for details). However, it does not traverse the usual control flow graphs (CFGs) but abstracted versions, in which entire code fragments matching specific patterns are collapsed into individual *nodes*. Since the patterns can depend on the variables, separate abstracted CFGs must be constructed for each given hotvar. The construction is based on a straightforward syntax-directed algorithm as for example described in [14].[1] The only variation is that the algorithm first matches the program against the different patterns, using the algorithm described in the section above, and in the case of a match constructs a single node of the class corresponding to the successful pattern, rather than using the standard construction and recursively descending into the statements subterms.

In addition to the syntactic classes representing the different statement types of the programming language, the abstracted CFG can thus contain nodes of several different pattern classes. The algorithm requires *use-* and *def*-nodes and uses *barrier-*, *barrier-block-* and *block*-nodes as optimizations. All of these represent code chunks that the algorithm regards as opaque (to different degrees) because they contain no definition for the given variable. They can therefore be treated as atomic nodes for the purpose of path search, which drastically reduces the number of paths that need be explored. *barrier*-nodes represent any statements that require annotations, i.e., principally loops. They must therefore be re-expanded and traversed during the annotation of the algorithm. In contrast, *block*-nodes are completely irrelevant to the hotvar because they neither require annotations (i.e., contain no barriers) nor contribute to annotations (i.e., contain no occurrence of the hotvar in an *lvar*-position). They can thus also remain atomic during the annotation phase, i.e., are not entered on path traversal. Blocks are typ-

[1] Since the generators only produce well-structured programs, a syntax-directed graph construction is sufficient. However, if necessary, we could replace the graph construction algorithm by a more general version that can handle ill-structured programs.

ically loop-free sequences of assignments and (nested) conditionals. *barrier-blocks* constitute a further optimization by combining the other two concepts: they are essentially barriers wrapped into larger blocks. Hence, they must be re-expanded during annotation, like normal *barrier*-nodes. The algorithm must further distinguish between reaching a (barrier) block from behind and from within. Coming from behind, it can treat the block opaquely, as described above. Coming from within (i.e., starting from the initial use), the algorithm must ignore the block label, and regard the node as the underlying statement. This means it has to keep track of the previous location as it navigates along paths.

4.5 Annotation of Paths

For each use of a hotvar, the path computation in the previous section returns a list of paths to *putative* definitions: although they have been identified by successful matches, there is no way to tell at this stage which, if any, of the definitions are relevant. In fact, it may be that several separate definitions are needed to fully define a variable for a single use. In a sense, the paths are untrusted and their correctness is established by annotating all barriers between the uses and definitions. Since this must take control flow into account, the current annotation is computed as the weakest precondition of the previous annotation.

Paths are then annotated in two stages. First, unless it has already been done (during a previous path), the definition at the end of the path is annotated, and the current annotation is set to its postcondition (cf. Section 4.6). If the use is contained within the definition then the path does not need to be continued because the definition will have been fully annotated "internally"; otherwise, we go on to the rest of the path.

The path annotation (Figure 6) works back along the path from a use to a definition, computing weakest preconditions along the way, and annotating loops and barriers as appropriate. Both the computation of preconditions and the insertion of annotations are done node by node rather than statement by statement.

At each point, we know the current weakest precondition, the previous location, the original use location (i.e., the start of the current path) and the hotvar. The previous location is needed to compute the precondition, and the hotvar and use location are used to prevent duplicate annotations.

It first checks whether the current node is visible[2] from the definition. If so, then we are finished since the VCG will have all the information it needs from this point onwards. Likewise, if this is the last node (that is, the one before the def), then we're finished annotating. If not, we look to see if this node has already been annotated.

[2]A node is visible from another node if it comes after it in a path through the CFG and there are no barriers between the nodes.

```
proc ann_path(HotVar, Path, PrevLoc, Post, UseLoc) :=
  case Path of
    [] -> done
    Node::NodeList ->
    if node_visible(NodeList) or NodeList = [] then
      done
    else
      Loc := get_node_location(Node);
      NextNode := head(NodeList);
      NextLoc := get_node_location(NextNode);
      if is_annotated(Loc, Post, UseLoc, HotVar) then
        skip
      else
        if node_is_barrier_or_opaque(Node) then
          if within(PrevLoc, Loc) then
            if node_is_loop(Node) then
              if within(NextLoc, Loc) then
                ann_loop_node(Node, Post, UseLoc, HotVar)
              else
                ann_barrier_node(Node, Post, UseLoc, HotVar)
            else
              skip
          else ann_barrier_node(Node, Post, UseLoc, HotVar)
        else
          if node_is_loop(Node) then
            if within(NextLoc, Loc) then
              ann_loop_node(Node, Post, UseLoc, HotVar)
            else
              ann_barrier_node(Node, Post, UseLoc, HotVar)
          else skip;
        Pre := node_precondition(PrevLoc, Post, Node);
        ann_path(HotVar, NodeList, Loc, Pre, UseLoc)
```

Figure 6. Path Annotation Algorithm

If so, we skip to the next node. If not, we distinguish several cases, depending on whether it's a loop or a barrier or an *opaque* node (blocks and barrier blocks), whether the previous node is within the current node, and whether the next node is within the current node. Once we've dealt with a node, the weakest precondition of that node is calculated, and we move on to the next node.

The WPC of a node is somewhat subtle and depends on whether or not it is a barrier or opaque, the statement itself (for basic blocks), and the previous location. In many cases the WPC does not change. For those cases where it does, the new WPC needs to be computed by looking at the statement. We distinguish atomic and compound statements. Compound statements (series, if, for, while) can only change the WPC if the previous location is after a loop, in which case $\mathrm{WPC}(P, C) = \mathrm{end}(C) \Rightarrow P$, where P is the incoming postcondition, C is the statement, and $\mathrm{end}(C)$ is the end condition for the loop, $\bar{C}$. For **while** b **do** c, this is $\neg b$, and for **for** $i := e_1$ **to** e_2 **do** c, this is $i > e_2$. In other words, the WPC says "if the loop has terminated then P". For atomic statements we compute the weakest precondition by calling the VCG and simplifying the result.

4.6 Annotation of Nodes

The path traversal described above calls the actual annotation routines when it needs to annotate a node. For annotation, we distinguish three classes of nodes: definitions, barriers, and loops (i.e., basic nodes which are loops).

The most important (and interesting) class is the definitions. This is really the core of the whole system, and where the annotation knowledge is represented in the form of *annotation schemas*, which take a match (identifying the pattern and location), and use meta-programming to construct and insert the annotations.

For example, each initialization block from Figure 1 is defined by a separate pattern and has a corresponding annotation schema. In each case, a final outer postcondition $\forall I : 1 \leq I \leq n . \forall J . 1 \leq J \leq m . X_{\text{init}}(I, J) = init$ (where X is the matrix) is inserted, while 1(b) and 1(c) also get an inner postcondition, as well as inner and outer invariants.

Note that even after a pattern has been successfully matched, an annotation schema might still fail its preconditions. For example, the binary assignment schema (Figure 1(a)) simply matches against a sequence of assignments, but the schema further requires that the indices of the first and last assignments are the low and the high, respectively.

The annotation schemas can handle more complicated examples than the "pure" definitions directly reflected by the patterns. A common situation is for a barrier to appear within a definition. Consider the following simple example:

```
1  for i:=1 to N do
2    a[i]:=0;
3    for j:=1 to N do ...
```

The definition pattern is a single nested initialization, but the **for**-loop at (3) means that an extra postcondition, $a_{\texttt{init}}[i]$ = `init`, is needed on (2) to push the initialization through the body. However, if the **for**-barrier is *before* the assignment no extra annotation is needed. In general, the schemas are able to deal with such cases and maintain the "internal" flow of information within a definition.

5 Experiences

We have implemented the generic inference algorithm in about 4000 lines of documented Prolog code and instantiated it to certify initialization safety for code generated by AUTOBAYES and AUTOFILTER. The "declarative content" was surprisingly small: it only required instantiations of the pattern library but no changes to the algorithm itself.

5.1 AutoFilter

For AUTOFILTER, the definitions are given by the idioms in Figure 1 (both for vectors and matrices), along with the direct vector/matrix assignment operation ::=. This is captured by the following pattern:

$$
\begin{aligned}
def_{AF}(x) ::= \; & x := \_ \;||\; x ::= \_ \\
& ||\; (x[\_] := \_)+ \;||\; (x[\_,\_] := \_)+ \\
& ||\; \textbf{for}\, i := \_ \,\textbf{to}\, \_ \,\textbf{do}\, x[i] := \_ \\
& ||\; \textbf{for}\, i := \_ \,\textbf{to}\, \_ \,\textbf{do} \\
& \quad \textbf{for}\, j := \_ \,\textbf{to}\, \_ \,\textbf{do}\, x[i,j] := \_ \\
& ||\; \textbf{for}\, i := \_ \,\textbf{to}\, \_ \,\textbf{do} \\
& \quad \textbf{for}\, j := \_ \,\textbf{to}\, \_ \,\textbf{do} \\
& \quad \textbf{if}\, \_ \,\textbf{then}\, x[i,j] := \_ \,\textbf{else}\, x[i,j] := \_
\end{aligned}
$$

Like all patterns here, this is parametrized over a hotvar x, so that $def_{AF}(x)$ is the pattern of definitions for x, $barrier(x)$ (see below) is a barrier on a path from a use of x to its definition, and so on. Note that i and j are "free" meta-variables that get instantiated by the actual loop index variables. The patterns can also contain "junk", i.e., arbitrary code that can be interspersed with the match. This is easily defined by a junk operator omitted here.

Barriers are defined as **for**-loops without any occurrence of the hotvar. Loops *with* the hotvar are then simply treated by the normal CFG-routines, i.e., not collapsed. Finally, blocks are conditionals whose branches are deemed "irrelevant", which means they have no occurrence of a barrier or hotvar.

$$
\begin{aligned}
barrier_{AF}(x) ::= \; & x \notin (\textbf{for}\, \_ \,\textbf{to}\, \_ \,\textbf{do}\, \_) \\
block_{AF}(x) ::= \; & \textbf{if}\, (x \notin \_) \,\textbf{then}\, irr(x) \,\textbf{else}\, irr(x) \\
& ||\; \textbf{for}\, \_ \,\textbf{to}\, \_ \,\textbf{do}\, irr(x)
\end{aligned}
$$

where $irr(x) = (x \;||\; barrier_{AF}(x)) \notin \_$ is an auxiliary pattern blocking all occurrences of the hotvar or a barrier. We omit the easy pattern for uses.

5.2 AutoBayes

AUTOBAYES has similar patterns to AUTOFILTER, but does not need the ::= -pattern since it does not generate matrix operations, nor the assignment sequence pattern. It has two additional language constructs, **abort**, which appears in the definition pattern, and **while**-loops, which can form additional barriers. Blocks and uses are the same as for AUTOFILTER.

$$
\begin{aligned}
def_{AB}(x) ::= \; & \textbf{for}\, i := \_ \,\textbf{to}\, \_ \,\textbf{do}\, x[i] := \_ \\
& ||\; \textbf{for}\, i := \_ \,\textbf{to}\, \_ \,\textbf{do} \\
& \quad \textbf{for}\, j := \_ \,\textbf{to}\, \_ \,\textbf{do}\, x[i,j] := \_ \\
& ||\; \textbf{for}\, i := \_ \,\textbf{to}\, \_ \,\textbf{do} \\
& \quad \textbf{if}\, \_ \,\textbf{then abort else}\, x[i,j] := \_ \\
barrier_{AB}(x) ::= \; & x \notin (\textbf{for}\, \_ \,\textbf{to}\, \_ \,\textbf{do}\, \_) \\
& ||\; x \notin (\textbf{while}\, \_ \,\textbf{do}\, \_)
\end{aligned}
$$

Spec.	$\|P\|$	$\|A\|$	N	T_{gen}	T_{ATP}	$\|A\|$	N	T_{inf}	T_{ATP}
dsl	235	439	22/ -	16	41	494	19/ -	22	46
iss	523	441	27/ -	29	52	547	24/ -	46	49
segm	182	1278	105/ 6	22	628	1584	109/ -	54	202
	178	1332	114/10	24	903	1643	108/5	54	556

Table 1. Generated vs. Inferred Annotations

5.3 Results

Table 1 compares the results achieved by the new algorithm to those previously achieved in the certifiable code generation approach. The first two examples are AUTOFILTER specifications. `dsl` is taken from the attitude control system of NASA's Deep Space One mission [26]. `iss` specifies a component in a simulation environment for the Space Shuttle docking procedure at the International Space Station. `segm` describes an image segmentation problem for planetary nebula images taken by the Hubble Space Telescope. For this, AUTOBAYES synthesizes two different versions of an iterative numerical clustering algorithm. For each example, the table lists the size of the generated program, and then, for each approach, the sizes of the generated resp. inferred annotations, the numbers of generated and failed safety obligations, resp., as well as the runtimes and proof times in seconds.

For the two AUTOFILTER examples, both techniques prove to be very similar. The inferred annotations are slightly larger (by 15–25%) than the generated ones but, due to simplifications, they induce fewer VCs. For both approaches, the programs are certifiable fully automatically: all VCs are proven by the ATP. For the AUTOBAYES example, the situation is more complicated. Here, annotation generation has not kept up with ongoing development and the annotations are insufficient to prove the programs safe—even though they are. With the patterns described above, annotation inference can, in contrast, certify the first program but it too remains too weak for the second program, as a required code pattern is still missing. In both cases, the inferred annotations are again slightly larger, with fewer VCs induced.

Since it needs to build and traverse the CFGs, the inference approach is (substantially) slower than the generation approach, which only needs to expand templates. However, the introduction of *block*- and *barrier*-nodes cuts down the size of the CFGs dramatically, and we expect further speed-up from an optimized implementation. Moreover, the limiting factors overall are the proof times which are comparable (modulo failed VCs) in all cases, indicating that the inference does not introduce new complexity for the ATP.

6 Related Work

Logical annotations were recognized early on as one of the bottlenecks in program verification. Wegbreit [24] complained that "completely specifying the predicates on loops is tedious, error prone and redundant", and claimed that "loop predicates can be derived mechanically". Like other early work [9, 16], his approach is based on predicate propagation. Such methods use inference rules similar to a strongest postcondition calculus to push an initial logical annotation forward through the program. Loops are handled by a combination of different heuristics like weakening or strengthening and loop unrolling, until a fixpoint is achieved. However, these methods still need an initial annotation, and unlike our approach, the loop handling still induces a search space at inference time. Moreover, the constructed annotations are often only candidate invariants and need to be validated (or refuted) during inference, because they increase the search space.

Abstract interpretation has been used to infer annotations in separation logic for pointer programs [17] although the techniques required there are fairly specialized and elaborate compared to our patterns. The Coverity static analyzer [1] can be customized by macros that are simple versions of our patterns.

Finally, generate-and-test methods have been applied to our problem. Here, the generator phase uses a fixed pattern catalogue to construct candidate annotations while the test phase tries to validate (or refute) them, using dynamic or static methods. Daikon [10] is the best-known dynamic annotation inference tool in this category. Its tester accepts all candidates that hold without falsification but with a sufficient degree of support over the test suite. In order to verify the candidates, Daikon has also been combined [20] with the ESC/Java static checker [13]. In some cases, this combination even resulted in full safety proofs (wrt. the safety policy supported by ESC/Java). In general, however, dynamic annotation generation techniques remain incomplete because they rely on a test suite to generate the candidates and can thus miss annotations on paths that are not executed often enough (or not at all). Houdini [12] is a static generate-and-test tool that uses ESC/Java to statically refute invalid candidates. Since ESC/Java is a modular checker, Houdini has to start with a candidate set for the entire program and then iterate until a fixpoint is reached. This increases the computational effort required, and in order to keep the approach tractable, the pattern catalogue is deliberately kept small. Hence, Houdini is incomplete, and acts more as a debugging tool than as a certification tool.

7 Conclusions and Future Work

The certification system based on annotation inference as described here is much more flexible and extensible than our previous certification architecture [6]. Over time, extensions and modifications to our code generators had led to a situation of "entropic decay" where the generated annotations had not kept pace with the generated code. The new inference mechanism was able to automatically certify the same programs as the original system, as well as some subsequent extensions. However, as Table 1 shows, the reconstruction is not yet complete, and we continue to extend the new system. These system extensions require less effort than before since the patterns and annotation schemas are expressed declaratively and in one place, in contrast to the previous decentralized architecture where certification information is distributed throughout the code generator.

We have implemented several optimizations which cut down on redundant annotations. This is important since the same annotations can arise on multiple paths. Furthermore, many computational optimizations could be achieved by merging several of the phases.

Our approach offers a general framework for augmenting code generators with a certification component, and we have started a project to apply it to MathWorks Real-Time Workshop [2]. Our techniques could also be adapted to other annotation languages.

There is a strong interaction between the VCG and the annotations. It is possible to modify the VCG so that it does some analysis and requires less annotations. This would, however, mean that a greater part of the certification system must be trusted. Nevertheless, we would like have a "safety dial" whereby users can trade off trustedness with speed (which depends, *inter alia*, on the number of annotations which must be checked). Further empirical studies will be required to determine the most effective balance.

References

[1] www.coverity.com.

[2] www.mathworks.com/products/rtw/

[3] XML Path Language (XPath) Version 1.0, 1999. www.w3.org/TR/xpath.

[4] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming*. Addison-Wesley, 2005.

[5] E. Denney and B. Fischer. Correctness of source-level safety policies. In *FM'03*, *LNCS 2805*, pp. 894–913. Springer, 2003.

[6] E. Denney and B. Fischer. Certifiable program generation. In *GPCE'05*, *LNCS 3676*, pp. 17–28. Springer, 2005.

[7] E. Denney, B. Fischer, and J. Schumann. Adding assurance to automatically generated code. In *8th Intl. Symp. High-Assurance Systems Engineering*, pp. 297–299. IEEE Press, 2004.

[8] E. Denney, B. Fischer, and J. Schumann. An empirical evaluation of automated theorem provers in software certification. *Intl. J. of AI Tools*, 15(1):81–107, 2006.

[9] N. Dershowitz and Z. Manna. Inference rules for program annotation. *ICSE-3*, pp. 158–167. IEEE Press, 1978.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, 2001.

[11] B. Fischer and J. Schumann. AutoBayes: A system for generating data analysis programs from statistical models. *J. Functional Programming*, 13(3):483–508, 2003.

[12] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME'01*, *LNCS 2021*, pp. 500–517. Springer, 2001.

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI'02*, pp. 234–245. ACM Press, 2002.

[14] M.J. Harrold and G. Rothermel. Syntax-directed construction of program dependence graphs. Technical Report OSU-CISRC-5/96-TR32, The Ohio State University, 1996.

[15] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *JACM*, 50(1):63–69, 2003.

[16] S. Katz and Z. Manna. Logical analysis of programs. *CACM*, 19(4):188–206, 1976.

[17] O. Lee, H. Yang, and K. Yi. Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis. In *ESOP'05*, *LNCS 3444*, pp. 124–240. Springer, 2005.

[18] J. C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[19] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *PLDI'98*, pp. 333–344. ACM Press, 1998.

[20] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected invariants: Integrating Daikon and ESC/Java. In *First Workshop on Runtime Verification*, *Elec. Notes in Theoretical Computer Science*, 55(2). Elsevier, 2001.

[21] C. Rich and L. M. Wills. Recognizing a programs's description: A graph-parsing approach. *IEEE Software*, 7(1):82–89, 1990.

[22] D. R. Smith. KIDS: A semi-automatic program development system. *IEEE TSE*, 16(9):1024–1043, 1990.

[23] M. Stickel et al. Deductive composition of astronomical software from subroutine libraries. In *CADE-12*, *LNAI 814*, pp. 341–355. Springer, 1994.

[24] B. Wegbreit. The synthesis of loop predicates. *CACM*, 17(2):102–112, 1974.

[25] M. Whalen, J. Schumann, and B. Fischer. Synthesizing certified code. In *FME'02*, *LNCS 2391*, pp. 431–450. Springer, 2002.

[26] J. Whittle and J. Schumann. Automating the implementation of Kalman filter algorithms. *ACM Trans. Mathematical Software*, 30(4):434–453, 2004.